LES PROVISIONERS ET TAINTS

Introduction

Terraform nous aide non seulement à créer et à gérer les infrastructures, mais également à les approvisionner lors de la création ou de la suppression de ressources.

Pour cela, il utilise une fonctionnalité nommée `provisioner`, Les provisionneurs sont utilisés pour **exécuter des scripts ou des commandes shell sur une machine locale ou distante** dans le cadre de la création/suppression de ressources. Ils sont similaires aux user-data dans une instance EC2 qui ne s'exécutent qu'une seule fois lors de la création. D'ailleurs, Terraforme est même capable de les exécuter en cas d'échec.

Cependant je vous conseille d'utiliser les provisionneurs qu'en dernier recours. Pour la plupart des situations courantes, il existe de meilleures alternatives, comme l'utilisation d'Ansible (pour info, j'ai faits un cours complet sur Ansible), voire les services proposés directement par les fournisseurs d'infrastructures, comme nous avions pu le faire sur notre instance EC2 avec l'utilisation du `user_data`.

Les provisionneurs Terraform

Il existe de nombreux provisionneurs disponibles, nous allons cependant **étudier les provisionneurs Terraform les plus importants**.

Le provisionneur local-exec

Le provisionneur `local-exec` appelle un **exécutable local** après la création d'une ressource. Cela appelle un processus et non sur la ressource que nous créons.

Dans l'exemple ci-dessous, nous voulons écrire l'adresse IP publique de notre instance EC2 dans un fichier nommé `ip_address.txt` :

```
provider "aws" {
  profile    = "default"
  region     = "us-west-2"
}

resource "aws_instance" "my_ec2" {
  ami           = "ami-06ffade19910cbfc0"
  instance_type = "t2.micro"
}

provisioner "local-exec" {
    command = "echo ${aws_instance.my_ec2.public_ip} > ip_address.txt"
}
```

Nous spécifions alors un bloc `provisioner` en spécifiant le type de provisionneur à utiliser, dans pour notre exemple, nous utilisons le type `local-exec`.

D'autres options sont également prises en charge par ce type de provisionneur : :

- `working_dir` (Facultatif) : spécifie le répertoire de travail où la commande sera exécutée. Il peut être fourni comme chemin relatif vers le répertoire de travail en cours ou comme chemin absolu, sans oublier que le répertoire doit bien sûr exister.
- `interpreter` (Facultatif) : S'il est fourni, il s'agit d'une liste d'arguments d'interpréteurs utilisés pour exécuter votre commande. Le premier argument est l'interpréteur lui-même. Il peut être fourni comme chemin relatif vers le répertoire de travail en cours ou comme chemin absolu. Les arguments restants sont les options de l'interpréteur à lancer avant la commande (Ex : `interpreter = ["perl", "-e"]`). S'il n'est pas spécifié, les interpréteurs par défaut

seront choisis en fonction du système d'exploitation du système.

Exécutez la commande `terraform init && terraform apply` et observez la commande du provisionneur `local-exec` exécutée localement vos commandes sur votre machine exécutant Terraform. Vous pouvez ensuite afficher le contenu du fichier **public_ip.txt** pour vérifier l'enregistrement de votre adresse IP publique :

```
cat ip_address.txt
```

Résultat :

```
54.87.56.65
```

Ça reste le type de provisionneur le plus simple à utiliser car nous n'avons pas à nous soucier de spécifier des informations de connexion pour le moment.

Le provisionneur remote-exec Paire de clé ssh

le provisionneur `remote-exec` permet d' **appeler un script sur une ressource distante** une fois qu'elle est créée. Nous pouvons fournir une liste de chaînes de commandes qui sont exécutées dans l'ordre où elles sont fournies.

Pour utiliser un provisionneur remote-exec, vous devez choisir une connexion SSH ou WinRM dans un bloc de code nommé `connection` de votre ressource. L'exemple suivant est un peu spécifique à AWS, mais peut très facilement être adapté pour les autres fournisseurs, . Sur AWS, par défaut il n'est pas possible d'utiliser un mot de passe pour se connecter à son instance EC2, à la place cela se fait à travers un échange de clé ssh. Il faut donc au préalable **créer notre paire de clé ssh**.

Dans cet exemple, nous utiliserons le mode de connexion en SSH. Pour cela, si vous n'aviez pas déjà généré votre paire de clé ssh, alors commencez déjà par la créer sans phrase secrète et utilisez le nom terraform (ou autre) avec la commande suivante :

```
ssh-keygen -t rsa

Generating public/private rsa key pair.
Enter file in which to save the key (/home/hatim/.ssh/id_rsa): ./terraform
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Ensuite, mettez à jour les autorisations de votre clé privée avec la commande suivante :

```
chmod 400 ./terraform
```

À ce moment là, vous pouvez utiliser le provisionneur **`remote-exec`**, comme suit :

```
provider "aws" {
    region  = "us-west-2"
}

resource "aws_key_pair" "my_ec2" {
    key_name   = "terraform-key"
    public_key = file("./terraform.pub")
}

resource "aws_security_group" "instance_sg" {
  name = "terraform-test-sg"

    egress {
        from_port       = 0
        to_port         = 0
        protocol        = "-1"
        cidr_blocks     = ["0.0.0.0/0"]
    }

    ingress {
        from_port = 80
        to_port = 80
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
```

```
    ingress {
        from_port = 22
        to_port = 22
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }

}

resource "aws_instance" "my_ec2" {
    key_name      = aws_key_pair.my_ec2.key_name
    ami           = "ami-06ffade19910cbfc0"
    instance_type = "t2.micro"
    vpc_security_group_ids = [aws_security_group.instance_sg.id]
    connection {
        type        = "ssh"
        user        = "ubuntu"
        private_key = file("./terraform")
        host        = self.public_ip
    }

    provisioner "remote-exec" {
        inline = [
          "sudo apt-get -f -y update",
          "sudo apt-get install -f -y apache2",
          "sudo systemctl start apache2",
          "sudo systemctl enable apache2",
          "sudo sh -c 'echo \"<h1>Hello devopssec</h1>\" > /var/www/html/index.html'"
        ]
    }
}
```

Nous commençons par créer une ressource `aws_key_pair` requise pour les **connexions SSH sur notre instance EC2** nous spécifions d'abord le nom de la paire de clés ainsi que notre clé publique créée précédemment afin d'autoriser notre machine locale à se connecter sur notre instance ec2.

Ensuite, nous imbriquons le bloc `connection` en spécifiant qu'on souhaite se connecter avec le protocole ssh sur la machine cible avec l'utilisateur `ubuntu` (utilisateur par défaut sur les AMIs Ubuntu) et notre clé privée créée précédemment. Nous sommes également obligés de fournir l'adresse de la ressource à laquelle se connecter dans l'argument `host`, cependant les blocs `connection` ne peuvent pas faire référence à leurs ressources parent par leur nom.

Au lieu de cela, nous utilisons l'objet spécial `self` qui représente la ressource parent de la connexion et possède tous les arguments de cette ressource. Dans notre cas nous l'utilisons pour récupérer l'adresse IP publique de notre instance EC2.

Enfin nous indiquons qu'on souhaite utiliser le provisionneur `remote-exec` en spécifiant les commandes à exécuter sur la machine distante. Voici d'ailleurs deux autres options utiles pour ce type de provisionneur :

- `script` : il s'agit d'un chemin (relatif ou absolu) vers un script local qui sera copié sur la ressource distante puis exécuté. Ce dernier ne peut pas être fourni avec l'option `inline` ou `scripts`.
- `scripts` : Il s'agit d'une liste de chemins (relatifs ou absolus) vers des scripts locaux qui seront copiés dans la ressource distante puis exécutés. Ils sont exécutés dans l'ordre où ils sont fournis. Ce dernier ne peut pas être fourni avec l'option `inline` ou `scripts`.

Nom d'utilisateur et mot de passe

Pour utiliser un nom d'utilisateur et un mot de passe, il suffit tout simplement de les spécifier dans votre bloc de `connection`, comme suit :

```
resource "aws_instance" "my_ec2" {
    ...

    connection {
        type        = "ssh"
        user        = "ubuntu"
        password    = "mot de passe"
        host        = self.public_ip
    }

    provisioner "remote-exec" {
        ...
    }
}
```

Le provisionneur file

Le provisionneur file est utilisé pour **copier des fichiers ou des répertoires** de la machine exécutant Terraform vers la ressource nouvellement créée. Il prend également en charge les connexions de type SSH et WinRM. Voici un exemple d'utilisation :

```
provider "aws" {
    region  = "us-west-2"
}

resource "aws_key_pair" "my_ec2" {
    key_name   = "terraform-key"
    public_key = file("./terraform.pub")
}

resource "aws_instance" "my_ec2" {
    key_name      = aws_key_pair.my_ec2.key_name
    ami           = "ami-06ffade19910cbfc0"
    instance_type = "t2.micro"

    connection {
        type        = "ssh"
        user        = "ubuntu"
        private_key = file("./terraform")
        host        = self.public_ip
    }

    provisioner "file" {
        source      = "apache2.conf"
        destination = "/etc/apache2/apache2.conf"
    }

    provisioner "file" {
        content     = "Options All -Indexes"
        destination = "/var/www/html/.htaccess"
    }
}
```

Vous remarquerez que son utilisation reste très similaire au provisionneur `remote-exec`. Dans cet exemple j'ai utilisé différent arguments, que je vous explique en juste ci-dessous :

- `source` : il s'agit du fichier ou dossier source. Il peut être spécifié comme chemin relatif au répertoire de travail actuel ou comme chemin absolu. Cet attribut ne peut pas être spécifié avec l'option `content`.
- `content` : c'est le contenu à copier sur la destination. Si la destination est un fichier, le contenu sera écrit sur ce fichier, dans le cas d'un répertoire, un fichier nommé **tf-file-content** est créé, c'est pour cela qu'il est recommandé d'utiliser un fichier comme destination. Cet attribut ne peut pas être spécifié avec l'attribut `source`.
- `destination` (obligatoire) : il s'agit du chemin de destination. Il doit être spécifié comme chemin absolu.

provisionneurs de destruction

Il est également possible de définir des provisionneurs qui s' **exécutent uniquement pendant une opération de destruction**. Ils sont utiles pour effectuer le nettoyage du système, extraire des données, etc.

Pour lancer un provisionneur de destruction, il suffit de rajouter l'instruction `when = "destroy"`. Exemple :

```
resource "aws_instance" "my_ec2" {
    # ...

    provisioner "local-exec" {
        when    = "destroy"
        command = "echo 'destruction de l'instance ${aws_instance.my_ec2.public_ip}'
    }
}
```

Ressources Taints

Définition

Par défaut, les provisionneurs s'exécutent lorsque la ressource dans laquelle ils sont définis est créée. Les provisionneurs au moment de la création ne sont exécutés que pendant la création , pas pendant la mise à jour ou tout autre cycle de vie. Ils sont conçus comme un moyen d'effectuer un amorçage d'un système.

Si un provisionneur au moment de la création échoue, la ressource est marquée comme "Taint" (corrompue/contaminée) . Une ressource contaminée sera planifiée pour être détruite et recréée lors de la prochaine éxecution de la commande `terraform apply`. Terraform le fait car un provisionneur défaillant peut laisser une ressource dans un état semi-configuré. Parce que Terraform ne peut pas raisonner sur ce que fait le provisionneur, la seule façon de garantir la création correcte d'une ressource est de la recréer.

Ressources corrompue manuelle

Dans les cas où vous souhaitez **forcer la destruction et recréation manuelle d'une ressource**, Terraform a une commande spéciale nommée `terraform taint` et intégrée dans la CLI. Cette commande ne modifiera pas l'infrastructure, mais modifie le fichier d'état afin de marquer une ressource comme corrompue. Une fois qu'une ressource est marquée comme corrompue, le prochain plan montrera que la ressource sera détruite et recréée et la prochaine demande mettra en œuvre ce changement.

Pour corrompre une ressource, utilisez la commande suivante :

```
terraform taint resource.id
```

`resource` fait référence au nom du bloc de notre ressource et l'`id` fait référence à l'identifiant de la ressource à corrompre. Soit pour code suivant :

```
resource "aws_instance" "my_ec2" {
    ami           = "ami-06ffade19910cbfc0"
    instance_type = "t2.micro"
}
```

Nous utiliserons la commande suivante pour forcer la corruption de cette ressource :

```
terraform taint aws_instance.my_ec2
```

Comportement en cas d'échec

Par défaut, les provisionneurs qui échouent entraînent également la ressource en question à l'état d'échec. Pour modifier ce comportement , vous pouvez modifier le paramètre `on_failure`. Les valeurs autorisées sont :

- `continue` : ignorer l'erreur et poursuivre la création ou la destruction.
- `fail` (comportement par défaut) : générer une erreur et arrêter d'exécuter les prochaines instructions du code. S'il s'agit d'un provisionneur de création, la ressource sera corrompue.

```
resource "aws_instance" "my_ec2" {
    # ...

    provisioner "local-exec" {
        on_failure = "continue"
        command = "echo ${aws_instance.my_ec2.public_ip} > ip_address.txt"
    }
}
```

Conclusion

Nous avons fait le tour des provisionneurs que je juge les plus utiles, cependant je vous laisse le soin de découvrir d'autres types de provisionneurs qui s'adaptent selon des besoins spécifiques (provisionneur de type Chef, provisionneur de type Puppet, etc ...). Vous avez également appris à mieux comprendre et gérer les comportements de vos provisionneurs. Dans le prochain chapitre nous étudierons plus en détails les "states" (états) sur Terraform.